

Solving the TTC 2011 Compiler Optimization Case with GReTL

Dipl.-Inform. Tassilo Horn

horn@uni-koblenz.de

Institute for Software Technology
University Koblenz-Landau, Campus Koblenz

This paper discusses the GReTL solution of the TTC 2011 *Compiler Optimization* case [1]. The submitted solution covers both the *constant folding* task and the *instruction selection* task. The *verifier* for checking the validity of the graph is also implemented, and some additional test graphs are provided as requested by the *extension*.

1 Introduction

GReTL (Graph Repository Transformation Language, [5]) is the operational transformation language of the TGraph technological space [2]. Models are represented as typed, directed, ordered, and attributed graphs. GReTL uses the GReQL (Graph Repository Query Language, [3]) for its matching part.

The standard GReTL operations are currently targeted at out-place transformation which create the target metamodel and the target graph simultaneously, but this case requires changing the graph in-place. Luckily, GReTL is designed as an extensible language. Adding custom operations requires only specializing one framework class, overriding one method that implements the operation's behavior, and implementing another factory method that is responsible for creating an instance of the new operation initialized with the given parameters. Consequently, some in-place operations have been added to the language. There is one operation *MatchReplace* which is similar to a rule in graph replacement systems. There is an operation *Delete* for deleting elements, an operation *MergeVertices* for merging a set of duplicate vertices into one canonical representative, and a higher-order operation *Iteratively* which receives another operation and applies it as long as possible. The implementation of the most complex one, *MatchReplace*, is about 380 lines of Java code including comments and empty lines, the other three operations are at most 80 lines or less. Thus, this case was solved using a mixed bottom-up and top-down approach: the language was extended with three new operations, and then these operations were used to write the transformations.

2 Case Solutions

In this section, the two transformation tasks are discussed in sequence. The solutions can be run on the SHARE image [4].

2.1 Constant Folding

The GReTL solution for the constant folding task does a bit more than what was required. It contains operation calls that realize the following optimizations:

- (1) Binary operations depending only on two Const vertices are replaced with with a Const containing the result of evaluating the binary.
- (2) Not operations depending only on a Const are similarly replaced by the evaluation result.
- (3) Unused Const vertices are deleted.
- (4) Const vertices representing the same value are merged into one.
- (5) Cond nodes (conditional jumps) depending on a Const are replaced with unconditional jumps (Jmp).
- (6) Unreachable code is eliminated.
- (7) The operand edges of Phi vertices are renumbered if needed.
- (8) Phi vertices with only one argument are replaced by a direct dependency between the argument and the vertex depending on that Phi.
- (9) Block vertices containing only an unconditional Jmp are removed.
- (10) In subgraphs consisting of commutative and associative binary operations (Add or Mul) constants are pulled up in order to generate new possibilities for the constant folding optimization (1), e.g., a graph structure $((1 \star x) \star 2)$ is replaced with $((1 \star 2) \star x)$ for $\star \in \{+, *\}$.

In the following, the optimizations (1), (3), and (4) are discussed, starting with the evaluation of binary operations depending only on constants. A helper function is defined that receives some Binary operation bin and the values of its left and right operand (lval, rval) and returns the evaluation result.

```

1 evaluateBinary() := using bin, lval, rval:
2   hasType{Add}(bin) ? lval + rval
3   : hasType{Sub}(bin) ? lval - rval
4   : hasType{Mul}(bin) ? lval * rval
5   : hasType{Div}(bin) ? (let d := lval / rval in d < 0 ? ceil(d) : floor(d))
6   : hasType{Mod}(bin) ? lval % rval
7   : hasType{Shl}(bin) ? bitOp("SHIFT_LEFT", lval, rval)
8   : hasType{Shr}(bin) ? bitOp("UNSIGNED_SHIFT_RIGHT", lval, rval)
9   : hasType{Shrs}(bin) ? bitOp("SHIFT_RIGHT", lval, rval)
10  : hasType{And}(bin) ? bitOp("AND", lval, rval)
11  : hasType{Or}(bin) ? bitOp("OR", lval, rval)
12  : hasType{Eor}(bin) ? bitOp("XOR", lval, rval)
13  : hasType{Cmp}(bin) ?
14    (bin.relation = "GREATER" ? (lval > rval ? 1 : 0)
15    : bin.relation = "GREATER_EQUALS" ? (lval >= rval ? 1 : 0)
16    : bin.relation = "LESS" ? (lval < rval ? 1 : 0)
17    : bin.relation = "EQUAL" ? (lval = rval ? 1 : 0)
18    : bin.relation = "NOT_EQUAL" ? (lval <> rval ? 1 : 0)
19    : bin.relation = "LESS_EQUAL" ? (lval <= rval ? 1 : 0)
20    : bin.relation = "TRUE" ? 1
21    : bin.relation = "FALSE" ? 0
22    : error("Don't know how to handle " ++ bin.relation))
23  : error("Don't know how to handle " ++ bin);

```

Depending on the type of the given Binary bin, the correct operation is dispatched using a sequence of conditional expressions¹ and applied to the operand values. The division has to be handled specially: in GReQL, a division results in a double value, but here an integer division is intended. To match these semantics, either the floor or the ceiling of the result d is taken.

The next listing shows the replacement of all Binary vertices that have two Const arguments with a new Const with value set to the result of evaluating the binary operation. The MatchReplace operation has semantics similar but not identical to rules in graph replacement systems. Its first parameter is a template graph that describes the structure of the subgraph to be created or changed (similar to the RHS in rules in graph replacement systems), and following the arrow symbol, there is a GReQL query that reports a set of matches. The query is comparable to the LHS in graph transformations, except that it

¹GReQL conditional expression: <testExp> ? <trueExp> : <falseExp>

calculates all matches instead of one match at a time. For all matches, the template graph is applied, thereby skipping matches containing previously modified elements.

In the template graph, the dollar (\$) is a variable that holds the current match. Vertices are represented using parentheses and edges with arrows and curly braces:

```
(Type 'greql' | attr1 = 'greql2', attr2 = 'greql3')
-->{Type 'greql' | attr1 = 'greql2', attr2 = 'greql3'}
```

The semantics for vertices, but likewise for edges, are as follows. The optional Type is a vertex type name. If no Type is specified and greql evaluates to a vertex in the current match or any other vertex in the graph, it is bound to the template vertex and preserved. If Type is given and greql evaluates to a vertex in the current match, then that vertex is replaced with a new vertex of the specified type, all edges incident to the replaced vertex are relinked to the new one, and the new vertex is bound to the template vertex. If Type is given but no greql query, then a new vertex of the given type is created and bound to that template vertex. The attributes of the bound elements are set to the result of the queries in the attribute list. All vertices and edges in the match that are not bound to any template graph element are deleted. The deletion of some vertex implies deletion of all incident edges.

```
24 Iteratively MatchReplace ('startBlock') <-- {Dataflow | position = '-1'}
25 (Const '$.op' | value = '$.value')
26 <== from binOp: V{Binary}
27 with hasType{Const}(lconst) and hasType{Const}(rconst)
28 reportSet rec(op: binOp,
29             value: evaluateBinary(binOp, lconst.value, rconst.value),
30             edges: edgesFrom(binOp)) end
31 where argEdges := sortByPosition(nonContainmentDfs(binOp)),
32         lconst := endVertex(argEdges[0]),
33         rconst := endVertex(argEdges[1]);;
```

The higher-order operation Iteratively applies the MatchReplace operation as long as some match could be replaced. The query given to MatchReplace reports a set of matches where each match is a record of a binary operation (op), the evaluation result gathered by the evaluateBinary() helper (value), and the set of edges starting at the binary operation (edges).

The template graph specifies that a Const has to be created with value set to \$.value. The new Const should be connected to the StartBlock. The connecting edge has to be of type Dataflow with position = -1. Because the new Const refers to the binary operation op and additionally specifies a type, it means the replacement of that binary with the new constant. All edges incident to the binary will be relinked to the constant. Finally, the binary op and all edges starting at it will be deleted, because they occur in the match but not in the template graph.

This operation may have deleted vertices that were the single element depending on some Const. Thus, the following operation deletes all unused Const vertices.

```
34 Delete <== from c: V{Const} with inDegree(c) = 0 reportSet c end;
```

Furthermore, the operation may have created duplicate Const vertices, i.e., constants with the same value. The next operations merges them in order to have exactly one Const per needed value.

```
35 MergeVertices <== from const: V{Const}
36 reportMap const -> from d: V{Const}
37 with d.value = const.value and d <> const
38 reportSet d end
39 end;
```

The MergeVertices operation receives a query that has to result in a map assigning to vertices a set of duplicates that should be merged. Any Const vertex is mapped to the set of all different Const vertices with equal value. Thus, for any used value there will be exactly one canonical Const.

2.2 Instruction Selection

In this section, all operation calls for the instruction selection task are discussed. The first one transforms all Binary vertices with at least one Const argument to immediate target binary operations.

```

1 MatchReplace (#"Target" ++ typeName($.binary) ++ "I" ' $.binary ' | value = '$.value ' ,...)
2 <== from b: V{Binary}
3   with count(constEdges) > 0
4   reportSet rec(binary: b,
5     value: endVertex(constEdges[0]).value ,
6     edge: constEdges[0]) end
7   where constEdges := from e: edgesFrom{Dataflow}(b)
8     with (b.commutative = false ? e.position = 1 : e.position < -1)
9     and hasType{Const}(endVertex(e))
10    reportSet e end;

```

The query iterates over all Binary vertices that depend on at least one constant. The variable `constEdges` is bound to all Dataflow argument edges leading to a Const with one additional restriction: if the binary operation is not commutative, then the immediate operation has to select the right operand (`position = 1`). If the binary is commutative, then any operand may be used.

The query reports a set of records. In each record, `binary` refers to the Binary vertex, `value` refers to the value of the Const operand's value attribute, and `edge` refers to the the Dataflow edge that connects `binary` to the Const operand. The Const is deleted, and its value is pulled into the immediate operation.

The interesting point is that the template graph specifies that the binary has to be replaced with a new vertex whose type is specified by a query instead of being fixed. Here, the type name is calculated by concatenating "Target" with the original binary's type name followed by "I". Since both the type name and the reference query are optional, a type name specified as query is denoted with a leading `#`.

The value attribute is set to the value reported in each match record. The three dots (...) indicate that all other attributes that are equally defined for the type of the replaced binary and the new immediate binary have to be copied, e.g., the associative and commutative values. This saves one further operation call for transforming Cmp vertices which are usual binaries with an additional relation attribute.

The next operation replaces MemoryNode vertices (Load/Store) that reference a SymConst to immediate target load/store vertices.

```

11 MatchReplace (#"Target" ++ typeName($.memory) ++ "I" ' $.memory ' | symbol = '$.sym' , ...)
12 <== from m: V{MemoryNode} ,
13   df: edgesFrom{Dataflow}(m) ,
14   symConst: m --df-> & {SymConst}
15   reportSet rec(memory: m, sym: symConst.symbol , df: df) end;

```

Again, the query returns a set of records. The memory node `memory` is replaced by a new immediate target memory node whose `symbol` attribute is set to the value of the original SymConst's `symbol` value. The Dataflow edge `df` connecting the original memory node to the SymConst is deleted. Again, further attribute values are copied.

The two operations creating immediate target operations have created orphaned Const and SymConst vertices, i.e., vertices of these types that no other vertex depends on. Thus, they can be delete.

```

16 Delete <== from c: V{Const , SymConst} with inDegree(c) = 0 reportSet c end;

```

Finally, the transformation replaces all other vertices whose type has a target counterpart.

```

17 MatchReplace (#"Target" ++ typeName($) ' '$ ' | ...)
18 <== V{^TargetNode , ^TargetMemoryNode , ^TargetMemoryNode ,
19   ^Block , ^Argument , ^Start , ^End , ^Phi , ^Return , ^Sync};

```

The query returns the set of all vertices that are not of type `TargetNode`, `TargetMemoryNode`, `TargetMemoryNodeL`, `Block`, `Argument`, `Start`, `End`, `Phi`, `Return`, or `Sync`. The former three exclude already transformed vertices, and the rest are nodes of types that don't have a different target type.

Again, in the template graph the type of the new vertex is specified with a query. Since the match query evaluates to a set of vertices in contrast to a set of records, the variable `$` holding the current match is just a vertex which will be replaced while copying attribute values from the replaced to the new vertex.

3 Conclusion

In this paper, the parts of the constant folding and the complete instruction selection transformation were explained. With respect to the evaluation criteria, both the *Constant Folding* as well as the *Instruction Selection* solution are complete and correct, at least for the provided test graphs.

With respect to performance, all given graphs could be transformed in times between one and two seconds. However, running the constant folding transformation on the 100.000 nodes graph published shortly before the contest takes about six minutes. GReQL's set-based semantics of always calculating all matching elements has considerable influence especially on the `MatchReplace` operation. This operation skips matches containing elements of previous matches, because the previous application might have changed them in a way that they wouldn't match anymore. Thus, in the worst case any match except the first one has been calculated for nothing.

With respect to conciseness, about 190 lines of code including comments and empty lines for three helpers and twelve operation calls is about as much as one would expect for the constant folding transformation. In contrast, the instruction selection transformation with its 4 operation calls subsuming to 42 lines of code is very concise.

Concerning purity, the transformations are specified completely in GReTL. But the language has been extended with three in-place operations appropriate for this task. In this respect, both judging the solution as pure as well as impure can be justified. Nevertheless, it demonstrates GReTL's extensibility which is a strength in its own respect.

References

- [1] Sebastian Buchwald & Edgar Jakumeit (2011): *Compiler Optimization: A Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC 2011: Fifth Transformation Tool Contest*, Zürich, Switzerland, June 29-30 2011, EPTCS.
- [2] J. Ebert, V. Riediger & A. Winter (2008): *Graph Technology in Reverse Engineering, The TGraph Approach*. In R. Gimnich, U. Kaiser, J. Quante & A. Winter, editors: *10th Workshop Software Reengineering (WSR 2008)*, *GI Lecture Notes in Informatics* 126, GI, pp. 67–81.
- [3] Jürgen Ebert & Daniel Bildhauer (2010): *Reverse Engineering Using Graph Queries*. In: *Graph Transformations and Model Driven Engineering*, LNCS 5765, Springer, pp. 335–362, doi:10.1007/978-3-642-17322-6_15.
- [4] Tassilo Horn: *SHARE demo related to the paper Solving the TTC 2011 Compiler Optimization Case with GReTL*. http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu_10.04_TTC11_gretl-cases.vdi.
- [5] Tassilo Horn & Jürgen Ebert (2011): *The GReTL Transformation Language*. In Jordi Cabot & Eelco Visser, editors: *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, *Lecture Notes in Computer Science* 6707, Springer, pp. 183–197, doi:10.1007/978-3-642-21732-6_13.

A The Complete Solution

In this appendix, the complete GReTL source code for the optional verifier, the constant folding transformation, and the instruction selection transformation is printed.

A.1 The Verifier

The verifier is not part of the challenge, but especially when developing a transformation, a strict verification helps to spot errors. GReTL does not distinguish between libraries and transformations. Any transformation like Verifier (Listing 1) that does nothing except declaring helper functions can be considered a library. When executing such a transformation, the helpers are made available to the calling transformation.

```

1 transformation Verifier;
2
3 checkValidity() :=
4   count(V{Start}) = 1 and count(V{StartBlock}) = 1           // (1)
5   and count(V{End}) = 1 and count(V{EndBlock}) = 1          // (2)
6   and (forall df: E{Dataflow},                                // (3)
7       hasType{Block}(endVertex(df))
8       @ df.position = -1)
9   and (forall n: V{Node, ^Block}                               // (4)
10      @ exists! df: edgesFrom{Dataflow}(n)
11      @ df.position = -1 and hasType{Block}(endVertex(df)))
12   and (forall c: V{Const}                                       // (5)
13      @ c -->{Dataflow} theElement(V{StartBlock}))
14   and (forall                                                    // (6)
15      phi: V{Phi},
16      block: phi-->{Dataflow @ thisEdge.position=-1} & {Block}
17      @ outDegree{Dataflow}(phi) - 1 =                          // (6.1)
18      outDegree{Controlflow}(block)
19      and (forall                                                // (6.2)
20         posNo: list(0..outDegree{Controlflow}(block) - 1)
21         @ (exists! phiEdge: edgesFrom{Dataflow}(phi)
22            @ phiEdge.position = posNo)
23            and (exists! blockEdge:
24               edgesFrom{Controlflow}(block)
25               @ blockEdge.position = posNo)))
26   and (forall block: V{Block, ^EndBlock}                        // (7)
27      @ inDegree(block) > 0)
28   and (forall v: V @ degree(v) > 0);                           // (8)

```

Listing 1: The verifier as GReTL helper

In line 3, a GReQL helper named `checkValidity()` with no parameters is defined. It checks for eight constraints marked with numbered comments that must hold for the current graph.

- (1) There must be exactly one Start and one StartBlock.
- (2) There must be exactly one End and one EndBlock.
- (3) For all Dataflow edges pointing to a Block, the position attribute has to be set to -1.
- (4) For all Node vertices with the exception of Block vertices, there exists exactly one outgoing Dataflow edge with position set to -1 and leading to a Block, i.e., any node is contained in exactly one block.
- (5) All Const vertices are contained in the single StartBlock.
- (6) Phi vertices have the correct structure:
 - (6.1) Phi vertices have as many operand Dataflow edges as their containing Block has Controlflow predecessors.

- (6.2) For any position number between 0 and the number of Controlflow predecessors of the Phi's block minus one, there exists exactly one Phi argument edge with that position number and exactly one control flow predecessor with that position number.
- (7) There are no empty blocks.
- (8) There are no isolated vertices.

Because the helper is a conjunction of predicates, its result is a boolean value. The transformations discussed in the next sections are asserting that the helper returns true as first command (to ensure the given graph is correct) and as last command (to ensure the transformation produced a valid result).

A.2 The Constant Folding Transformation

In the following, the complete GReTL source code of the Constant Folding transformation is shown.

```

1 transformation ConstantFolding;
2
3 // "import" the checkValidity() helper
4 ExecuteTransformation "transforms/verifier.gretl";
5
6 Assert <== checkValidity();
7
8 // Bind the single StartBlock to a variable
9 startBlock := theElement(V{StartBlock});
10
11 // Evaluates the given Binary with lval and rval.
12 evaluateBinary() := using bin, lval, rval:
13   hasType{Add}(bin) ? lval + rval
14   : hasType{Sub}(bin) ? lval - rval
15   : hasType{Mul}(bin) ? lval * rval
16   : hasType{Div}(bin) ? (let d := lval / rval in d < 0 ? ceil(d) : floor(d))
17   : hasType{Mod}(bin) ? lval % rval
18   : hasType{Shl}(bin) ? bitOp("SHIFT_LEFT", lval, rval)
19   : hasType{Shr}(bin) ? bitOp("UNSIGNED_SHIFT_RIGHT", lval, rval)
20   : hasType{Shrs}(bin) ? bitOp("SHIFT_RIGHT", lval, rval)
21   : hasType{And}(bin) ? bitOp("AND", lval, rval)
22   : hasType{Or}(bin) ? bitOp("OR", lval, rval)
23   : hasType{Eor}(bin) ? bitOp("XOR", lval, rval)
24   : hasType{Cmp}(bin) ?
25     (bin.relation = "GREATER" ? (lval > rval ? 1 : 0)
26     : bin.relation = "GREATER_EQUALS" ? (lval >= rval ? 1 : 0)
27     : bin.relation = "LESS" ? (lval < rval ? 1 : 0)
28     : bin.relation = "EQUAL" ? (lval = rval ? 1 : 0)
29     : bin.relation = "NOT_EQUAL" ? (lval <> rval ? 1 : 0)
30     : bin.relation = "LESS_EQUAL" ? (lval <= rval ? 1 : 0)
31     : bin.relation = "TRUE" ? 1
32     : bin.relation = "FALSE" ? 0
33     : error("Don't know how to handle " ++ bin.relation))
34   : error("Don't know how to handle " ++ bin);
35
36 // Sorts the given dataflows by the value of their position attribute.
37 sortByPosition() := using dataflows:
38   let posMap := from i: list (0..count(dataflows) - 1)
39     reportMap dataflows[i].position -> dataflows[i] end,
40     sortedKeys := sort(keySet(posMap))
41   in from i: sortedKeys
42     reportSet posMap[i] end;
43
44 // A helper that returns all non-containment dataflows of a node. That means,
45 // dataflows where position is not -1.
46 nonContainmentDfs() := using node:
47   from e: edgesFrom{Dataflow}(node)
48   with e.position <> -1

```

```

49  reportSet e end;
50
51  // Perform the contained transformations as long as one was applicable
52  Iteratively
53    // Replace associative, commutative Binaries of form ((x + y) + 1) with ((x +
54    // 1) + y), that is, pull up constants.
55    Iteratively MatchReplace b1('$ .b1'), b2('$ .b2'),
56      b1 -->{'$.df1'} b2 -->{'$.df3'} ('$ .c'),
57      b2 -->{'$.df4'} ('$ .y'),
58      b1 -->{'$.df2'} ('$ .x')
59    <== from b1: V{Add, Mul},
60      df1, df2: nonContainmentDfs(b1),
61      b2: b1 --df1-> & {Add, Mul @ type(b1) = type(thisVertex)},
62      df3, df4: nonContainmentDfs(b2)
63    with df1 < df2 and df3 < df4
64      // Used only by b1 + the containment Dataflow
65      and inDegree{Dataflow}(b2) = 2
66      and hasType{Const}(endVertex(df2))
67      and not(hasType{Const}(x))
68    reportSet rec(b1: b1, b2: b2, x: x, c: theElement(b1 --df2->),
69      y: y, df1: df1, df2: df2, df3: df3, df4: df4) end
70    where x := theElement(b2 --df3->),
71      y := theElement(b2 --df4->);
72
73  // Replace Binary with 2 Consts with a single Const of the result. Do
74  // iteration here, cause that operation is likely to produce another match
75  // immediately.
76  Iteratively MatchReplace ('startBlock') <-- {Dataflow | position = '-1'}
77    (Const '$.op' | value = '$.value')
78  <== from binOp: V{Binary}
79    with hasType{Const}(lconst) and hasType{Const}(rconst)
80    reportSet rec(op: binOp,
81      value: evaluateBinary(binOp, lconst.value, rconst.value),
82      edges: edgesFrom(binOp)) end
83  where argEdges := sortByPosition(nonContainmentDfs(binOp)),
84    lconst := endVertex(argEdges[0]),
85    rconst := endVertex(argEdges[1]);
86
87  // Replace unary ops (i.e., Not) with the value
88  Iteratively MatchReplace ('startBlock') <-- {Dataflow | position = '-1'}
89    (Const '$.op' | value = '$.value')
90  <== from uniOp: V{Not}
91    with hasType{Const}(endVertex(argEdge))
92    reportSet rec(op: uniOp,
93      value: bitOp("NOT", endVertex(argEdge).value),
94      edges: edgesFrom(uniOp)) end
95  where argEdge := theElement(nonContainmentDfs(uniOp));
96
97  // Remove unused Consts
98  Delete <== from c: V{Const} with inDegree(c) = 0 reportSet c end;
99
100  // Merge duplicate Consts into one for each value.
101  MergeVertices <== from const: V{Const}
102    reportMap const -> from d: V{Const}
103      with d.value = const.value
104      and d < const
105      reportSet d end
106  end;
107
108  // Remove duplicate edges from StartBlock to Consts.
109  Delete <== from c: V{Const}
110    reportSet difference(dfs, set(dfs[0])) end
111  where dfs := edgesFrom{Dataflow}(c);
112
113  // Replace Cond nodes depending only on a Const with an unconditional Jmp.

```



```

114 MatchReplace ('$.block') <--{'$.df'} (Jump) <--{'Controlflow'} ('$.target')
115 <== from cond: V{Cond}
116   with not isEmpty(cond -->{'Dataflow'} & {'Const'})
117   reportSet rec(cond: cond, // the Cond
118     df: blockTup[0], // the Dataflow to the containing Block
119     block: blockTup[1], // the containing Block
120     cf: targetTup[0], // the True/False edge to the target
121     target: targetTup[1], // the target of the Cond
122     edges: edgesConnected(cond)) end
123   where const := theElement(cond -->{'Dataflow'} & {'Const'}),
124     blockTup := theElement(from e: edgesFrom{'Dataflow'}(cond)
125       with e.position = -1
126       reportSet e, endVertex(e) end),
127     targetTup := theElement(from e: edgesTo{'Controlflow'}(cond)
128       with const.value = 0 ? hasType{False}(e)
129         : hasType{True}(e)
130       reportSet e, startVertex(e) end);
131
132 // Now delete any node that has no outgoing Dataflow recursively
133 Iteratively Delete <== from node: V{^StartBlock}
134   with outDegree(node) = 0
135   reportSet node end;
136
137 // Remove the superfluous operand edges of Phi
138 Delete <== from phi: V{Phi}, opEdge: edgesFrom{'Dataflow'}(phi)
139   with opEdge.position < -1
140   and not(contains(predBlockCfPositions, opEdge.position))
141   reportSet opEdge end
142   where predBlock := theElement(phi -->{'Dataflow @ thisEdge.position = -1'}),
143     predBlockCfPositions := from df: edgesFrom{'Controlflow'}(predBlock)
144       reportSet df.position end;
145
146 // Renumber Phi operand edges and the corresponding block predecessor edges.
147 MatchReplace ('$.phiOperand') <--{'$.phiOpEdge' | position = '$.position'} ('$.phi'),
148   ('$.blockPred') <--{'$.blockCfEdge' | position = '$.position'} ('$.block')
149 <== from phi: V{Phi}, i: list(0..count(blockEdges) - 1)
150   reportSet rec(phiOperand: endVertex(phiOpEdges[i]),
151     phiOpEdge: phiOpEdges[i],
152     position: i,
153     phi: phi,
154     block: block,
155     blockCfEdge: blockEdges[i],
156     blockPred: endVertex(blockEdges[i])) end
157   where phiOpEdges := sortByPosition(nonContainmentDfs(phi)),
158     block := theElement(phi -->{'Dataflow @ thisEdge.position = -1'}),
159     blockEdges := sortByPosition(edgesFrom{'Controlflow'}(block));
160
161 // Remove Phi with only one argument
162 MatchReplace ('$.arg') <--{'$.depEdge'} ('$.dep')
163 <== from phi: V{Phi}
164   with count(argEdges) = 1
165   reportSet rec(arg: theElement(phi --argEdge--),
166     depEdge: userEdge,
167     dep: theElement(phi <-userEdge--),
168     phi: phi,
169     edges: edgesConnected(phi)) end
170   where argEdges := nonContainmentDfs(phi),
171     argEdge := argEdges[0],
172     userEdge := theElement(edgesTo{'Dataflow'}(phi));
173
174 // Remove Blocks that contain only a single Jump
175 MatchReplace ('$.pred') <--{'$.predEdge'} ('$.target')
176 <== from block: V{Block}, jmp: V{Jump}
177   with block <-{'Dataflow @ thisEdge.position = -1'} jmp
178   and inDegree(block) = 1

```

```

179     reportSet rec(pred: theElement(block -->{Controlflow}),
180                 predEdge: theElement(edgesFrom{Controlflow}(block)),
181                 target: theElement(jmp <--{Controlflow}),
182                 edges1: edgesConnected(block),
183                 edges2: edgesConnected(jmp),
184                 block: block) end;
185
186 ; // End outer Iteratively
187
188 // Finally, check if still everything's correct.
189 Assert <== checkValidity();

```

A.3 The Instruction Selection Transformation

In the following, the complete GReTL source code of the Instruction Selection transformation is shown.

```

1  transformation InstructionSelection;
2
3  // "import" and execute the checkValidity() helper
4  ExecuteTransformation "transforms/verifier.gretl";
5  Assert <== checkValidity();
6
7  // Transform all Binaries with at least one Const to an immediate TargetBinary
8  // operation.
9  MatchReplace (#""Target" ++ typeName($.binary) ++ "I" '$.binary' | value = '$.value', ...)
10 <== from b: V{Binary}
11     with count(constEdges) > 0
12     reportSet rec(binary: b,
13                 value: endVertex(constEdges[0]).value,
14                 edge: constEdges[0]) end
15 where constEdges := from e: edgesFrom{Dataflow}(b)
16     with (b.commutative = false ?
17         // For non-commutative binary ops, we have to chose the
18         // dataflow on position 1, i.e., the second argument
19         e.position = 1
20         // ... else anything except -1 is ok.
21         : e.position <> -1)
22     and hasType{Const}(endVertex(e))
23     reportSet e end;
24
25 // Transform Load/Store with SymConsts to TargetLoadI/TargetStoreI.
26 MatchReplace (#""Target" ++ typeName($.memory) ++ "I" '$.memory' | symbol = '$.sym', ...)
27 <== from m: V{MemoryNode},
28     df: edgesFrom{Dataflow}(m),
29     symConst: m --df-> & {SymConst}
30     reportSet rec(memory: m, sym: symConst.symbol, df: df) end;
31
32 // Delete unused Const/SymConst vertices.
33 Delete <== from c: V{Const, SymConst} with inDegree(c) = 0 reportSet c end;
34
35 // Transform the rest of the elements. TargetMemoryNode and TargetMemoryNodeI
36 // have to be excluded explicitly, because they are not derived from TargetNode.
37 MatchReplace (#""Target" ++ typeName($) '$' | ...)
38 <== V{^TargetNode, ^TargetMemoryNode, ^TargetMemoryNodeI,
39     ^Block, ^Argument, ^Start, ^End, ^Phi, ^Return, ^Sync};
40
41 // Finally, check if still everything's correct.
42 Assert <== checkValidity();

```